

# Apunte 05 - Programación Orientada a Objetos en Java

## Introducción a POO

De la misma forma que hemos dicho que la presente asignatura no pretende ser un curso de Java, decimos ahora que tampoco es nuestro objetivo se un curso de Programación Orientada a Objetos, en el caso del Lenguaje Java ya hemos explicado que proponemos revisar los elementos fundamentales para poder implementar los elementos de backend propuestos. En el caso de Programación Orientada a Objetos, sin embargo, entendemos que es un tema ya abordado en materias anteriores o concurrentes con Backend de Aplicaciones y por lo tanto nos limitaremos a revisar las herramientas de Java para implementar la POO agregando solo algunos comentarios sobre buenas prácticas.

Lo primero que nos gustaría, mostrar en este apunte y es uno de los por qué de la POO en lo que respecta al manejo de la complejidad, para eso vamos a tomar como ejemplo guía del presente apunte el siguiente caso:

Vamos a realizar una comparación elemental de complejidad entre el cálculo del promedio de 10 números enteros y el cálculo del promedio de 10 fracciones tratando de aproximar la complejidad a partir de las líneas de código y sin entrar en mayor detalle como sería por ejemplo un análisis de complejidad ciclomática.

Si pensamos en el código de del cálculo del promedio de 10 números aleatorios tendríamos algo como lo que sigue:

```
11          // Primero con números enteros
12          System.out.println("Primero con enteros");
13          int acumuladorI = 0;
14          for(int i = 0; i < 10; i++)
15          {
16              int aux = (int) (Math.random() * 10);
17              System.out.println("Generado: " + aux);
18              acumuladorI += aux; // acumuladorI = acumuladorI + aux;
19          }
20
21          double mediaI = acumuladorI / 10.0;
22          System.out.println("\n\nLa media es: " + mediaI);
```

Es evidente que un análisis de complejidad del ejemplo anterior indicaría que el problema es trivial y prácticamente no tiene complejidad apreciable. Sin embargo, que pasaría si queremos hacer lo mismo pero con fracciones, es decir obtener la fracción (numerador / denominador) promedio de 10 fracciones (numerador / denominador) generadas con valores aleatorios. Es evidente que la complejidad sería diferente al menos en la versión básica apoyada en las herramientas elementales del lenguaje Java. Desde las simples

operaciones de suma y división requeridas para el cálculo del promedio a la necesidad en el caso de las fracciones de simplificar para no obtener una fracción con números enormes.

La idea entonces es programar una clase fracción, que resuelva que permita crear objetos de tipo fracción y que resuelva las operaciones existentes ya para los números enteros en java y, de esta forma, lograr aproximar la complejidad de los problemas en esencia iguales pero que sin embargo, no son iguales como discutimos previamente.

## Primeros pasos

Para hacer eso, los lenguajes orientados a objetos (como Java) usan descriptores de entidades conocidos como *clases*. Básicamente, una *clase* es la descripción de una entidad u objeto de forma que luego esa descripción pueda usarse para crear muchos objetos que respondan a la descripción dada. Para establecer analogías, se puede pensar que una clase se corresponde con el concepto de *tipo de dato* de la programación estructurada tradicional, y los objetos creados a partir de la clase (llamados *instancias* en el mundo de la *POO*) se corresponden con el concepto de *variable* de la programación tradicional. Así como el *tipo* es uno solo y describe la forma que tienen todas las muchas *variables* de ese tipo, la *clase* es única y describe la forma y el comportamiento de los muchos *objetos* de esa clase.

Para describir objetos que responden a las mismas características de forma y comportamiento, se declara una *clase*. La definición de una clase incluye esencialmente dos elementos:

- **Atributos:** Son *variables* que se declaran dentro de la clase, y sirven para indicar la *forma* de cada objeto representado por esa clase. Los atributos, de alguna manera, muestran lo que cada objeto *es*, o también, lo que cada objeto *tiene*.
- **Métodos:** Son funciones, procedimientos o rutinas declaradas dentro de la clase, usados para describir el *comportamiento* de los objetos descriptos por esa clase. Los métodos, de alguna manera, muestran lo que cada objeto *hace*.

Como sabemos, una aplicación o programa Java típico debe incluir un método especial, llamado *main()*. Ese método es el primero que se ejecuta cuando se pide lanzar la aplicación, y desde allí se administra la participación de cada *instancia* u *objeto* creado.

Al desarrollar una aplicación en Java, el programador creará por lo general varias clases que luego se usarán a su vez para crear objetos que trabajarán juntos. Es común en ese sentido declarar una clase cuyo único objetivo sea contener al método *main()*. En el contexto de nuestro curso, eso es lo que hemos hecho y seguiremos haciendo mediante nuestra convención de declarar una clase con el nombre *App* o el que se convenga, para que sea ella la que contenga a *main()*. No obstante, digamos que *main()* podría estar incluido en cualquier clase, y que incluso cada clase del proyecto podría tener un *main()*. No es obligatorio que una clase se llame *App*, y que esta deba contener a *main()*. Es sólo una convención de trabajo en el curso.

La definición de una clase en Java se hace mediante la palabra reservada *class*, seguida de un par de llaves que delimitan su contenido. Es común (pero no obligatorio) que los atributos de la clase se declaren antes que los métodos. El conjunto de atributos y métodos de una clase se conoce como el conjunto de *miembros* de la clase. En nuestro ejemplo vamos a requerir la clase Fracción para resolver en ella los problemas asociados a una fracción, pero, que luego permita la creación de tantas fracciones concretas como haga falta en el programa:

```
3 public class Fraccion
4 {
5     // Atributos (Datos miembro)
6     private int numerador;
7     private int denominador;
8 }
```

Los atributos de la clase son variables, que pueden ser de tipo primitivo o pueden ser objetos de otras clases. De hecho, *String* es una clase de Java y no un tipo primitivo, por lo cual si algún atributo de la clase *Fraccion* fuera un *String*, tendríamos un atributo que es un objeto de otra clase, y así los objetos comienzan a colaborar entre ellos... Para la clase *Fraccion*, mínimamente sería necesario contar con un atributo que indique el numerador y otro que indique el denominador. La se vería como la que mostramos en la imagen anterior.

Uno de los principios de la POO es el llamado *Principio de Ocultamiento*, por el cual se sugiere que al definir una clase, no se permita que los atributos sean accesibles en forma directa desde el exterior de la clase, sino que se usen métodos de la misma para consultar sus valores o modificarlos. La idea detrás del *Principio de Ocultamiento* es que el programador que use una clase predefinida no deba preocuparse por los detalles de implementación interna de la clase, sino que simplemente use los métodos que la misma provee y se maneje en un nivel de abstracción más alto. Así, si usamos objetos de la clase *String*, no debemos preocuparnos por cómo está representada esa cadena dentro del objeto. Sabemos que dentro hay una cadena implementada de alguna forma convincente, y la clase brinda todos los métodos para hacer lo que necesitemos hacer. Además, el Principio de Ocultamiento permite que la clase controle qué valores tiene cada atributo y de qué forma esos valores deberían cambiar. Si se permite el acceso a un atributo de la clase desde fuera de ella, podría provocarse que un valor incorrecto, no validado, sea asignado en ese atributo haciendo que desde allí en adelante algún proceso interno de la clase falle, por ejemplo si permitimos que el denominador de la fracción sea cero...

## Modificadores de Acceso

La forma que Java provee para que un programador obligue a respetar el Principio de Ocultamiento son los llamados *modificadores de acceso*. Se trata de ciertas palabras reservadas que colocadas delante de la declaración de un atributo o de un método de una clase, hacen que ese atributo o ese método tengan accesibilidad más amplia o menos amplia desde algún método que no esté en la clase. Así, los calificadores de acceso en Java pueden ser cuatro: *public*, *private*, *protected*, y "*default*" (este último no es una palabra reservada: es el estado en el que queda un miembro si no se antepone ninguno de los otros tres calificadores anteriores):

- **public**: un miembro público es accesible tanto desde el interior de la clase (por sus propios métodos), como desde el exterior de la misma (por métodos de otras clases).
- **private**: sólo es accesible desde el interior de la propia clase, usando sus propios métodos.
- **protected**: aplicable en contextos de herencia (tema que veremos más adelante), hace que un miembro sea público para sus clases derivadas y para clases en el mismo paquete, pero los hace privados para el resto. ☐!!!

- "**default**" : un miembro que no sea marcado con ningún calificador de acceso, asume estatus de acceso *por defecto*, lo cual significa que será público para clases en el mismo paquete, pero privado para el resto. Notar (otra vez) que la palabra "default" en realidad no es una palabra reservada, sino sólo el nombre del estado en que queda el miembro.

Por todo lo expresado, es que en la clase *Fraccion* se han declarado los atributos como *private*. Entre los métodos de una clase, el más característico es el llamado método *constructor*. Un constructor es un método cuyo objetivo básico es el de inicializar los atributos de un objeto en el momento en que ese objeto o instancia se crea. Como veremos en breve, un objeto se crea en Java usando un operador del lenguaje llamado *new*, y el constructor se invoca *automáticamente* al crear con *new* una instancia de la clase.

Un constructor lleva el mismo nombre que la clase que lo contiene. No se debe indicar ningún tipo devuelto para él (ni siquiera *void*), y puede recibir parámetros como un método normal. Por otra parte, tanto los constructores como cualquier otro método de la clase pueden ser *sobrecargados*. Eso significa que se pueden definir *varias versiones del mismo método*. El compilador distingue entre las diversas sobrecargas de un método *por la forma de su lista de parámetros*. Por lo tanto, si un método tiene varias versiones definidas en una clase, las distintas versiones deben diferir en la cantidad de parámetros, en el tipo de los parámetros, o en ambas características. Notar que el cambio en el tipo devuelto por el método no define una nueva sobrecarga: solo las formas diversas en la lista de parámetros. En base a esto, la clase *Fracción* podría tener este aspecto si agregamos algunos constructores:

```
3  public class Fraccion
4  {
5      // Atributos (Datos miembro)
6      private int numerador;
7      private int denominador;
8
9      // Constructores
10     new *
11     public Fraccion(int num, int den)
12     {
13         numerador = num;
14         setDenominador(den);
15     }
16
17     new *
18     public Fraccion(int num)
19     {
20         this(num, den: 1);
21     }
22
23     new *
24     public Fraccion(Fraccion aCopiar) {
25         this(aCopiar.numerador, aCopiar.denominador);
26     }
27
```

Notar que en primero de los constructores la lista de parámetros está compuesta de dos números enteros, que son utilizados para inicializar el numerador y el denominador, también se puede observar que para el caso del denominador estamos usando un comportamiento propio de la clase que revisaremos en el siguiente apartado.

En el segundo caso tenemos un constructor que recibe un solo número entero y entienden en este caso la inicialización de una fracción impropia asignando en el denominador un 1. En este ejemplo también descubrimos el uso de la palabra reservada *this* que es la referencia implícita al propio objetos para invocación de métodos y atributos, y en el caso que aquí se puede observar el llamado explícito a otro constructor. La restricción para hacer este llamado es que debe realizarse si o sí como primera línea de código de un constructor y no puede ser utilizada en ningún otro lado para invocación de constructores.

Finalmente la tercera sobrecarga del constructor recibe como parámetro un objeto de la clase Fracción y provoca una nueva facción con los mismos valores de numerador y denominador que la fracción dada. En el apartado siguiente veremos la utilidad de este constructor y su necesidad para sortear la situación de la asignación de referencias, más allá de que en java el proceso correcto para provocar esta copia se denomina clonación y será analizado en clases posteriores.

Ahora bien, ahora tenemos una clase que es capaz de contener los datos del numerador y denominador de una fracción pero no tenemos aún una razón de ser para la misma, es decir no tenemos ningún método que provoque un comportamiento real, es decir que sea capaz de responder a un mensaje propio del concepto de una fracción.

Vamos a agregar 2 métodos extra en la clase *Fraccion* antes de comenzar a utilizar objetos de tipo fracción, por un lado el comportamiento más elemental de cualquier fracción que es devolver el valor real de la relación y en segundo lugar un comportamiento que como veremos más adelante toda clase java puede implementar para transformar la instancia de un objeto de la clase en una cadena de caracteres. Con estos métodos nuestra clase Fracción agregaría:

```
new *  
84     public double valorReal()  
85     {  
86         double resp = numerador / (double) denominador;  
87         return resp;  
88     }  
89  
90     new *  
91 ⚡↑     public String toString()  
92     {  
93         return "[" +this.numerador + "/" + this.denominador + "]";  
94     }  
95
```

## Referencias y Creación de Objetos

Así como está definida, la clase *Fraccion* está todavía muy incompleta pero ya puede usarse para crear objetos y mostrar la forma básica de usar un constructor. Supongamos entonces que se quieren crear dos o tres objetos de la clase *Fraccion* para comenzar a trabajar con ellos. El método *main()* de nuestra clase *App* podría hacerlo:

```

5 ► public static void main(String[] args)
6 {
7     //Declaro una referencia a Fraccion
8     Fraccion f1 = null;
9
10    // Instancio el objeto.
11    // f1 = new Fraccion(); // Utiliza el constructor por defecto y no compila porque al agregar constructor con
12    // parámetros el constructor por defecto deja de existir
13    System.out.println(f1);
14
15    // f1.numerador = 3; // compila porque los atributos están definidos com privados.
16    // f1.denominador = 5;
17
18    // System.out.println("\tValor real: " + f1.valorReal()); // provocaría un error ya que aún no creamos el objeto.
19
20    f1 = new Fraccion( num: 2, den: 3); // creamos la fracción 2 tercios
21    System.out.println(f1); // al imprimir invocamos el método toString, notar que se invoca de manera implícita
22    System.out.println(f1.valorReal()); // mostramos el valor real de la fracción al usar el comportamiento para eso
23
24    System.out.println("Fin!");
25
26 }

```

Note el uso del operador *new* para crear los objetos: ese operador crea un objeto de la clase que se le indique, cada vez que se lo usa. Pero en cada creación, *new* llama a algún constructor que figure en la clase del objeto que se pide crear, y la decisión de cuál constructor llamar depende de la lista de parámetros actuales que se pase a ese constructor al invocarlo: puede verse que al crear el objeto *f1*, se está invocando al primero de los constructores pues la lista de parámetros actuales es de dos números enteros.

En síntesis: el primer *new* crea un objeto que será manejado por la variable *f1*, y en ese objeto los atributos *numerador* y *denominador* quedan valiendo los valores { 2, 3} respectivamente.

Una vez que se han creado objetos de una clase (por ejemplo, lo que vimos en el método *main()*), estos objetos pueden comenzar a *invocar métodos* para que se apliquen sobre sus atributos. La forma de hacerlo, consiste en usar la variable que maneja al objeto y *colocar luego de ella un punto, seguido del nombre del método* que se quiere invocar para ese objeto como podemos observar al utilizar el método *valorReal* de la fracción referenciada por *f1*.

Se dice que el objeto desde el cual se invoca al método está *llamando* al método, o también se dice que a ese objeto se le está *pasando un mensaje*. En el ejemplo, el objeto *f1* está llamando al método *valorReal()* (o también decimos que *f1* recibe el mensaje de retornar su valor real). La salida del ejemplo anterior sería:

```

↑ "C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\JetBra
↓ null
→ [2/3]
→ 0.6666666666666666
→ Fin!

```

## Métodos de acceso de lectura y escritura

Sigamos entonces con la clase *Fraccion*. Si los atributos de una clase se definen privados, entonces la clase puede proveer métodos que permitan el acceso al valor de esos atributos, tanto para consultar su valor como para modificarlo (si es decisión del programador de la clase permitir esas operaciones) notar que al implementar estos métodos estamos diseñando el comportamiento de la clase. Por ejemplo en la clase fracción no podemos permitir que se modifiquen de manera independiente el numerador o el denominador

puesto que al modificar uno y solo uno de los atributos en realidad estaríamos en presencia de una nueva fracción y por lo tanto de un nuevo objeto aunque estos es conceptual.

La especificación de Java Beans en Java define que para esos métodos intervengan las palabras "get" (para los métodos de consulta) y "set" para los métodos modificadores, junto al nombre del atributo al que se quiere proveer acceso con la primera letra en mayúscula. Así, si un atributo se llama "clave", el método de consulta para el mismo podría llamarse "getClave()" y el modificador "setClave()". En la clase *Fraccion*, incluimos ahora este conjunto de *métodos de acceso*.

```
new ^  
27     public int getNumerador() {  
28         return numerador;  
29     }  
30  
31     new *  
32         public int getDenominador() {  
33             return denominador;  
34         }  
35  
36     new *  
37         private void setDenominador(int den) {  
38             if (den != 0)  
39                 denominador = den;  
40             else  
41                 denominador = 1;  
42         }  
43
```

Note que los métodos de consulta de la clase se ha marcado como *public* (incluso los constructores). Esto es consecuencia directa de respetar el *Principio de Ocultamiento*, que en última instancia aconseja declarar privados a los atributos y públicos a los métodos de una clase. Los métodos de acceso son muy sencillos: cada uno de los métodos tipo *get* retorna el valor del atributo con el cual se asocia, el método tipo *setDenominador* cambia el valor de ese atributo teniendo en cuenta el control a realizar. No hemos implementado el método *setNumerador* ya sería privado al igual que *setDenominador* y que no hay control necesario y podemos desde dentro de la clase acceder directamente al atributo. Cabe aclarar que en algunos equipos se conviene en programar siempre todos los métodos *set* aunque no cumplan función alguna y solo para mantener coherencia.

Finalmente, digamos que toda clase Java en última instancia hereda o se define a partir de otra muy general llamada *Object*, la cual provee ya definidos una serie de métodos elementales. Varios de esos métodos se

usan tal como vienen desde *Object*, pero algunos deberían ser redefinidos por cada clase. El método *toString()* es uno de ellos, y se usa para retornar una cadena de caracteres con el contenido del objeto invocante, de forma que sea adecuadamente visualizable en un dispositivo de salida. Si no se redefine, el método *toString()* retorna una cadena con el nombre de la clase a la cual pertenece el objeto, más la dirección de memoria de ese objeto en formato hexadecimal. En general, nuestras clases deberían contar con una versión propia del método *toString()*, lo cual es normalmente fácil de hacer. Mostramos la clase *Fraccion* completa con ese método incluido al final:

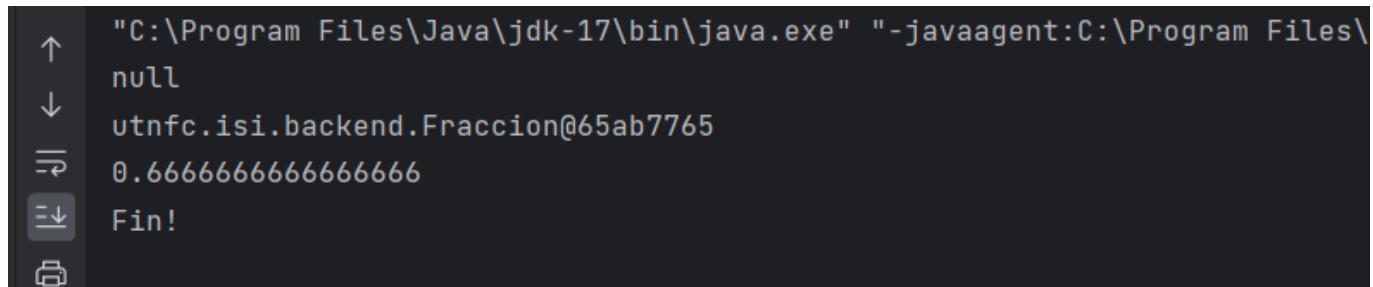
El método *toString()* es muy especial en Java. El lenguaje asume *automáticamente* que se está invocando a ese método en cualquier contexto en el cual se use un objeto pero se requiera una conversión a *String* del mismo. Eso significa que la línea:

```
System.out.println("Fracción f1: " + f1.toString());
```

también puede escribirse así, y el resultado es exactamente el mismo:

```
System.out.println("Fracción f1: " + f1);
```

Pero aún menos intuitivo es lo que pasa si en la clase fracción comentamos el método *toString* o le cambiamos el nombre de alguna forma. La salida anterior pasaría a ser como sigue:



```
"C:\Program Files\Java\jdk-17\bin\java.exe" "-javaagent:C:\Program Files\utnfc.isi.backend.Fraccion@65ab7765
null
0.6666666666666666
Fin!
```

Lo que está pasando aquí es que al cambiar el nombre del método en la clase *Fracción* el método *toString* que está usando java para obtener la versión de cadena de caracteres del objeto es el existente en la clase *Object* y por lo tanto muestra solo los elementos disponibles dentro de la clase *object* que son el nombre de la clase y la dirección de memoria de la referencia.

## Volviendo al promedio de 10 fracciones

Bueno, habiendo llegado hasta aquí tenemos nuestra clase *Fracción* con los métodos que ya hemos analizado y solo nos resta agregar los demás métodos a la clase para permitir que una fracción se pueda sumar, restar, multiplicar o dividir por otra fracción además de la posibilidad de simplificarse. La versión de fracción que acompaña el presente material ya tiene estos métodos implementados.

Ahora nos proponemos analizar la complejidad del problema planteado al inicio pero contando con la clase *Fracción* creada:

```
25
26     // Ahora con Fracciones
27     System.out.println("Ahora con fracciones");
28     Fraccion acumuladorF = new Fraccion( num: 0 );
29     for(int i = 0; i < 10; i++)
30     {
31         // ...
32
33         Fraccion aux = new Fraccion((int)(Math.random() * 7 + 1), (int)(Math.random() * 9 + 1));
34
35         System.out.println("Generado: " + aux);
36         acumuladorF = acumuladorF.sumarA(aux);
37     }
38
39
40     Fraccion mediaF = acumuladorF.dividirPor( num: 10 );
41     System.out.println("\n\nLa media es: " + mediaF);
42
43 }
44 }
```

Como podemos observar fuera de la necesidad de 2 números aleatorios en lugar de uno para la creación de cada fracción la complejidad entre este fragmento y el inicial asociado números enteros es similar puesto que ya hemos resuelto las problemáticas específicas de las fracciones dentro de la clase fracción.

[!Note]

### Objetos en base a Objetos

Acompañando este apunte además del ejemplo de fracciones que hemos analizado, agregamos un ejemplo más donde unos > objetos se crean en base a otros objetos más simples, vale su análisis para comprender el manejo de la complejidad por > capas que es lo que en general se persigue cuando realizamos diseños orientados a objetos.

## ❖ Reducción de Código Repetitivo con Lombok (Boilerplate Reduction)

En el desarrollo de aplicaciones con Java, uno de los problemas más comunes es la cantidad de código repetitivo que se debe escribir en las clases, incluso para tareas simples como declarar atributos, generar constructores, **getters**, **setters**, o redefinir **toString()** y **equals()**.

A este tipo de código, que es necesario pero no aporta lógica de negocio, se lo llama **boilerplate**. No es exclusivo de Java, pero en Java es especialmente visible.

En este contexto, la comunidad de Java ha desarrollado herramientas para reducir ese tipo de código repetitivo, y una de las más populares y aceptadas es **Lombok**.

### ¿Qué es Lombok?

**Lombok** es una biblioteca para Java que permite eliminar la necesidad de escribir código repetitivo mediante **anotaciones**. Con solo agregar una anotación sobre una clase (**@Getter**, **@Setter**, **@AllArgsConstructor**, **@ToString**, etc.), se genera automáticamente el código correspondiente en tiempo de compilación.

Este comportamiento es transparente: el código generado no aparece en los archivos **.java**, pero sí queda disponible en los archivos **.class**, y por lo tanto puede usarse como si estuviera escrito a mano.

### ¿Cómo funciona Lombok internamente?

Lombok se basa en un mecanismo conocido como **procesamiento de anotaciones (annotation processing)**, un estándar de Java que permite analizar y modificar el código durante el proceso de compilación. Para eso:

1. Utiliza una librería interna que intercepta el compilador (**javac**) y modifica el árbol de sintaxis abstracta (AST) del código fuente.
2. Inyecta automáticamente el código necesario según las anotaciones utilizadas.
3. El bytecode generado por el compilador incluirá getters, setters, constructores, etc., aunque no estén escritos explícitamente.

Este proceso ocurre **en tiempo de compilación**, por esto antes del tiempo de compilación el código que escribimos va a estar haciendo referencia a porciones de código inexistentes esto hace que también necesitemos extensiones de desarrollo y requiere que el IDE o entorno de compilación esté correctamente configurado para reconocer y aplicar las anotaciones de Lombok.

## ¿Cómo se configura Lombok?

Para poder utilizar Lombok:

1. **Agregar la dependencia** en el proyecto (por ejemplo en **pom.xml** si usás Maven):

```
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.30</version>
    <scope>provided</scope>
</dependency>
```

2. **Configurar el IDE** (IntelliJ, VS Code, Eclipse, etc.) para que reconozca Lombok:

- Por ejemplo, en IntelliJ: instalar el plugin Lombok desde el Marketplace.
  - Asegurarse de que “Enable annotation processing” esté activado en los ajustes del proyecto.
- O en VS Code no hace falta nada especial porque el soporte para lombok ya está incluido en el Java Extension Pack

## Explorando las principales anotaciones de Lombok

A continuación, presentamos un conjunto de anotaciones que ofrece Lombok para reducir el código repetitivo (boilerplate) en Java. Para cada una mostramos su utilidad, un ejemplo básico y cuál sería el equivalente manual.

### @Getter y @Setter

Estas anotaciones generan automáticamente los métodos **get...()** y **set...()** para los atributos de una clase.

```
import lombok.Getter;
import lombok.Setter;

@Getter
@Setter
public class Persona {
    private String nombre;
    private int edad;
}
```

### 👉 Equivalente manual:

```
public class Persona {
    private String nombre;
    private int edad;

    public String getNombre() { return nombre; }
    public void setNombre(String nombre) { this.nombre = nombre; }

    public int getEdad() { return edad; }
    public void setEdad(int edad) { this.edad = edad; }
}
```

También se pueden aplicar a nivel de atributo para más control.

#### 1. ¿Se pueden usar @Getter y @Setter por atributo en lugar de toda la clase?

Lombok permite aplicar sus anotaciones tanto a nivel de clase como a nivel de atributo, y esto es útil cuando:

- Si queremos exponer sólo algunos atributos (e.g., exponer getId() pero ocultar el resto).
- Si queremos que un atributo sea de sólo lectura o escritura.
- Si queremos aplicar una anotación específica sin que afecte toda la clase.

### ❖ Ejemplo:

```
public class Persona {

    @Getter
    private String nombre;

    @Getter @Setter
    private int edad;

    private String clave; // sin getter ni setter
}
```

⌚ Resultado:

- Se genera `getNombre()`
- Se genera `getEdad()` y `setEdad(...)`
- No se genera ningún método para clave

```
public class Persona {  
    private String nombre;  
    private int edad;  
  
    public String getNombre() { return nombre; }  
    public void setNombre(String nombre) { this.nombre = nombre; }  
  
    public int getEdad() { return edad; }  
    public void setEdad(int edad) { this.edad = edad; }  
}
```

**@ToString**

Genera automáticamente el método `toString()` incluyendo todos los atributos no `static` ni `transient`.

```
import lombok.ToString;  
  
@ToString  
public class Producto {  
    private String nombre;  
    private double precio;  
}
```

⌚ Equivalente manual:

```
public class Producto {  
    private String nombre;  
    private double precio;  
  
    public String toString() {  
        return "Producto(nombre=" + nombre + ", precio=" + precio + ")";  
    }  
}
```

**@NoArgsConstructor y @AllArgsConstructor**

Generan automáticamente:

- Un constructor vacío (`@NoArgsConstructor`)
- Un constructor con todos los atributos (`@AllArgsConstructor`)

```
import lombok.NoArgsConstructor;
import lombok.AllArgsConstructor;

@NoArgsConstructor
@AllArgsConstructor
public class Usuario {
    private String email;
    private String clave;
}
```

☞ Equivalente manual:

```
public class Usuario {
    private String email;
    private String clave;

    public Usuario() {}

    public Usuario(String email, String clave) {
        this.email = email;
        this.clave = clave;
    }
}
```

## @EqualsAndHashCode

Genera `equals()` y `hashCode()` considerando todos los atributos.

Este comportamiento puede modificarse agregando el parámetro (`of = "nombreAtributo"`) a la anotación

```
import lombok.EqualsAndHashCode;

@EqualsAndHashCode
public class Documento {
    private String tipo;
    private int numero;
}
```

☞ Equivalente manual (simplificado):

```
public class Documento {  
    private String tipo;  
    private int numero;  
  
    public boolean equals(Object o) {  
        if (this == o) return true;  
        if (o == null || getClass() != o.getClass()) return false;  
        Documento doc = (Documento) o;  
        return numero == doc.numero && tipo.equals(doc.tipo);  
    }  
  
    public int hashCode() {  
        return Objects.hash(tipo, numero);  
    }  
}
```

## @Data

Ahora la cosa se pone interesante, hasta acá vimos configuraciones independientes de cada aspecto y eso es completamente viable, sin embargo, ahora veremos configuraciones que incluyen a las anteriores, en este caso `@Data` genera automáticamente:

- `@Getter` y `@Setter`
- `@ToString`
- `@EqualsAndHashCode`
- `@RequiredArgsConstructor` (Incluye solo los atributos `final` o marcados como `@NonNull`)

Ideal para POJOs donde queremos todos los métodos básicos, por ejemplo:

```
import lombok.Data;  
  
@Data  
public class Alumno {  
  
    private final int legajo;  
  
    @NonNull  
    private String nombre;  
  
    private int idCurso;  
}
```

☞ Equivalente manual (simplificado):

```
import java.util.Objects;

public class Alumno {

    private final int legajo;
    private String nombre;
    private int idCurso;

    // Constructor requerido (legajo y nombre)
    public Alumno(int legajo, String nombre) {
        if (nombre == null) {
            throw new NullPointerException("nombre is marked non-null but is null");
        }
        this.legajo = legajo;
        this.nombre = nombre;
    }

    // Getter para legajo (no hay setter porque es final)
    public int getLegajo() {
        return legajo;
    }

    // Getter y Setter para nombre
    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        if (nombre == null) {
            throw new NullPointerException("nombre is marked non-null but is null");
        }
        this.nombre = nombre;
    }

    // Getter y Setter para idCurso
    public int getIdCurso() {
        return idCurso;
    }

    public void setIdCurso(int idCurso) {
        this.idCurso = idCurso;
    }

    // equals() y hashCode() considerando todos los campos
    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Alumno)) return false;
        Alumno alumno = (Alumno) o;
        return legajo == alumno.legajo &&
```

```

        idCurso == alumno.idCurso &&
        Objects.equals(nombre, alumno.nombre);
    }

    @Override
    public int hashCode() {
        return Objects.hash(legajo, nombre, idCurso);
    }

    // toString()
    @Override
    public String toString() {
        return "Alumno(legajo=" + legajo +
            ", nombre=" + nombre +
            ", idCurso=" + idCurso + ")";
    }
}

```

## @Builder

Permite generar un patrón Builder automáticamente.

```

import lombok.Builder;

@Builder
public class Cliente {
    private String nombre;
    private String direccion;
    private int edad;
}

```

👉 Uso:

```

Cliente c = Cliente.builder()
    .nombre("Juan")
    .direccion("Av. Siempreviva 123")
    .edad(40)
    .build();

```

Este patrón es especialmente útil cuando hay múltiples campos opcionales o con valores por defecto debido a la clase builder que se genera, en este punto no profundizamos el tema pero lo podemos sumar en ejemplos más adelante.

## @Value

Anotación para clases inmutables:

- Hace los atributos final
- Clase final
- Genera constructor, getters, `equals()`, `hashCode()` y `toString()`

```
import lombok.Value;

@Value
public class Punto {
    int x;
    int y;
}
```

Este último ejemplo agrega un concepto que hoy java introdujo en el lenguaje de programación a través de la declaración de `record`

## 📋 Anexo I: `@Value` de Lombok vs `record` de Java

¿Qué es un `record` en Java?

Un `record` es una **nueva estructura de clase inmutable**, pensada específicamente para modelar **objetos portadores de datos (data carriers)** de manera **concisa, segura, y legible**.

💡 Un `record`:

- Genera automáticamente:
  - `constructor`
  - `getters` (sin el prefijo `get`)
  - `equals()`, `hashCode()`, `toString()`
- Hace los campos `final` e inmutables
- No permite herencia (pero sí puede implementar interfaces)

❖ **Sintaxis:**

```
public record Alumno(int legajo, String nombre, int idCurso) {}
```

¿Qué hace `@Value` en Lombok?

```
import lombok.Value;

@Value
public class Alumno {
    int legajo;
    String nombre;
    int idCurso;
}
```

## @Value en Lombok:

- Declara automáticamente todos los atributos como `private final`
- Genera:
  - Constructor con todos los campos
  - Getters (`getNombre()`, etc.)
  - `equals()`, `hashCode()`, `toString()`
- Hace la clase `final` (no se puede extender)
- Requiere Lombok y configuración de anotaciones en el proyecto

## Comparativa detallada

Característica	@Value de Lombok	record de Java
Inmutabilidad	<input checked="" type="checkbox"/> (con <code>final</code> y sin setters)	<input checked="" type="checkbox"/> (todos los campos <code>final</code> por diseño)
Generación de métodos	Constructor, getters, equals, etc.	Constructor, <i>accessors</i> , equals, etc.
Herencia	No se puede extender (es <code>final</code> )	No permite herencia
Dependencia externa	Requiere Lombok	Parte del JDK desde Java 16
Accesores ( <code>getX()</code> )	Sí (estilo JavaBeans)	No ( <code>nombre()</code> en lugar de <code>getNombre()</code> )
Campos <code>private</code>	Sí	No → campos son públicos de solo lectura
Anidamiento	Admite clases internas normales	Requiere clases <code>static</code> anidadas
Personalización parcial	Sí (constructores, validaciones)	Sí (compact constructors, validations)
Patrón DTO / POJO	Muy usado	Ideal reemplazo moderno

## ¿Cuándo usar uno u otro?

Necesidad	Recomendación
Proyecto con Lombok ya integrado	Usá <code>@Value</code>
Proyecto Java moderno (16+) sin deps	Usá <code>record</code>
Necesitás compatibilidad anterior	<code>@Value</code> (Java 8+)
APIs públicas estilo JavaBeans	<code>@Value</code> (con getters)
Modelado de datos puro, simple	<code>record</code>

## Ejemplo completo comparado

### Con `@Value`

```
@Value
public class Alumno {
    int legajo;
```

```
String nombre;
int idCurso;
}
```

## Con record

```
public record Alumno(int legajo, String nombre, int idCurso) {}
```

## Uso

```
Alumno a = new Alumno(123, "Sofía", 4);
System.out.println(a.nombre()); // en record
System.out.println(a.getNombre()); // en @Value
```

## Conclusión

Si iniciamos de cero una aplicación con las últimas versiones de java, los **record** son, quizás, la **alternativa oficial** a las clases inmutables típicas. Son más livianos, integrados y expresivos.

Si trabajás con **versiones anteriores** o necesitás **estilo JavaBeans (get/set)** por herencia de dependencias o implementación de frameworks existentes, **@Value** sigue siendo una excelente opción, especialmente en proyectos con Lombok y Spring.

En la cátedra Backend de Aplicaciones hemos decidido optar por usar lombok para la reducción del boilerplate por lo que los ejemplos de la cátedra se implementarán en base a esa estructura.

## 📝 Refactorización completa de la clase Fracción usando Lombok

A modo de cierre de este apunte, proponemos volver a construir la clase Fracción que utilizamos para hacer el seguimiento de los conceptos básicos de la implementación del paradigma de programación orientado a objetos en Java. En esta oportunidad, la reimplementaremos utilizando Lombok, con el objetivo de ilustrar cómo esta herramienta permite reducir el boilerplate sin perder claridad en el diseño, y al mismo tiempo reforzar los conceptos fundamentales de encapsulamiento, constructores, validaciones y comportamiento orientado a objetos.

En este documento se presenta una **implementación completa y equivalente** de la clase **Fraccion** construida inicialmente de forma manual, pero ahora **aprovechando las anotaciones de Lombok**. Se busca demostrar cómo se puede lograr el mismo resultado funcional con menos código repetitivo, conservando legibilidad, buenas prácticas y validaciones.

---

### Parte 1 - Reducción básica con **@Data**

```
import lombok.Data;

@Data
public class Fraccion {
    private int numerador;
    private int denominador;
}
```

Esta versión básica con `@Data` reemplaza la necesidad de:

- `getNumerador()`, `getDenominador()`
- `setNumerador()`, `setDenominador()`
- `toString()`, `equals()`, `hashCode()`

Pero **todavía le faltan**:

- Validaciones del constructor
- Cálculo de MCD y simplificación
- Métodos de operaciones (suma, resta, etc.)

## Parte 2 - Versión completa con comportamiento equivalente

```
import lombok.Data;
import lombok.NonNull;

@Data
public class Fraccion {
    private int numerador;
    private int denominador;

    public Fraccion(int numerador, @NonNull int denominador) {
        if (denominador == 0) {
            throw new ArithmeticException("El denominador no puede ser cero");
        }
        this.numerador = numerador;
        this.denominador = denominador;
        simplificar();
    }

    public double valorReal() {
        return (double) numerador / denominador;
    }

    private int mcd(int a, int b) {
        return b == 0 ? a : mcd(b, a % b);
    }

    private void simplificar() {
        int divisor = mcd(Math.abs(numerador), Math.abs(denominador));
    }
}
```

```
        numerador /= divisor;
        denominador /= divisor;
    }

    public Fraccion sumar(Fraccion otra) {
        return new Fraccion(
            this.numerador * otra.denominador + otra.numerador * this.denominador,
            this.denominador * otra.denominador);
    }

    public Fraccion restar(Fraccion otra) {
        return new Fraccion(
            this.numerador * otra.denominador - otra.numerador * this.denominador,
            this.denominador * otra.denominador);
    }

    public Fraccion multiplicar(Fraccion otra) {
        return new Fraccion(
            this.numerador * otra.numerador,
            this.denominador * otra.denominador);
    }

    public Fraccion dividir(Fraccion otra) {
        if (otra.numerador == 0) {
            throw new ArithmeticException("No se puede dividir por una fracción
con numerador cero");
        }
        return new Fraccion(
            this.numerador * otra.denominador,
            this.denominador * otra.numerador);
    }
}
```

## ⌚ Análisis y comparación

- `@Data` se encarga de los métodos de acceso, comparación y representación textual.
- El constructor se define manualmente para incluir validación del denominador.
- Se mantiene la lógica de simplificación como parte del constructor.
- Se agregan los métodos de operaciones: `sumar`, `restar`, `multiplicar` y `dividir`.

🎓 Esta clase puede ahora utilizarse en tests, estructuras de colecciones, o para ejemplificar refactorización con herramientas modernas del ecosistema Java.

Esto permite **cerrar el ciclo de aprendizaje** desde lo manual y detallado hacia lo productivo y moderno.